6502 TDD und CI

- 6502 CPU und Assembler
- TDD – Test Driven Development
- CI – Continuous Integration

FAT32

- Die Welt will eine 6502 FAT32-Implementierung
  - die Spec.-konform ist
  - die mit allen Varianten/Geometrien „klar kommt“
  - die performant ist
  - „self contained“ - braucht nur read/write block

… challenge accepted!

FAT32

- Die Realität
  - FAT32-Implementierung „seit Jahren“ (2015)
  - kaum Fortschritt
  - Frustration
  - keine Motivation
  - Wir träumen von „Cluster-Chain-Support“

FAT32

Es muss sich was ändern, sonst stirbt das Steckschwein :(

ASM, Tests und CI

- Warum?
  - Ich spreche nur schlecht 6502
  - sehr schön akademisch ;)

ASM, Tests und CI

- Warum wirklich?
  - steckOS ist „komplex“ geworden
  - erfordert Modularisierung von Code
  - Produktivität
  - Test Driven Development
  - Regression Testing
  - Code Quality

ASM, Tests und CI

- Womit?
  - ca65/ld65 – Assembler/Linker (https://cc65.github.io/)
  - Py65 – Simulator/Monitor (https://github.com/mnaberez/py65)
  - Jenkins – Build-Server (https://jenkins.io/)

ASM, Tests und CI

- Vorraussetzungen, Was brauchen wir?
  - Entkopplung des Codes von konkreter Hardware
  - Test-API
  - Mocks

ASM, Tests und CI

- Was brauchen wir konkret?
  - assertA <expect> - vgl. Akku mit <expect>
  - assertX <expect> - vgl. X-Reg mit <expect>
  - assertY <expect> - vgl. Y-Reg mit <expect>
  - assert8, assert16, assert32 <expect> <address> - vgl. Bytes an Adresse mit <expect> (big endian)
  - assertCarry <0|1> - vgl. Carry mit <expect>
  - assertZero <0|1> - vgl. Zero mit <expect>
  - assertString <expect> <address> - ...
  - assertOut <expect> - vgl. expect mit char output
  - fail <message> - fail mit Meldung

ASM, Tests und CI

- Zeig doch mal, wie geht das!

```
;
;  hexout a binary number
;
.export hexout

.import char_out

hexout:
        pha
        phx

        tax
        lsr
        lsr
        lsr
        lsr
        jsr hexdigit
        txa
        jsr hexdigit
        plx
        pla
        rts

hexdigit:
        and #$0f        ;mask lsd for hex print
        ora #'0'        ;add "0"
        cmp #'9'+1      ;is it a decimal digit?
        bcc _out        ;yes! output it
        adc #6          ;add offset for letter a-f
_out:
        jmp char_out
```

```
.include "asmunit.inc"  ; unit test api

.import hexout      ; uut

.code
    lda   #$7e
    jsr   hexout

    assertOut "7E" ; assert outpuz
    assertA $7e    ; assert A is not destroyed

    lda   #$e7
    jsr   hexout

    assertOut "E7" ; assert outpuz
    assertA $e7    ; assert A is not destroyed

    lda   #$9f
    jsr   hexout

    assertOut "9F"
    assertA $9f

    brk

.segment "ASMUNIT"
```

ASM, Tests und CI

- Zeig doch mal, wie geht das!

```
$ (cd steckos/tools/lib/test/;make test_hexout ;cat test_hexout.bin.log)
ca65 --include-dir ../../../../asmunit --include-dir ../../../asminc --include-dir ../../../kernel -Dasmunit_char
_out=0x200 test_hexout.asm
ld65 --config ../../../../asmunit/asmunit.cfg ../toollib.a ../../../../asmunit/asmunit.a -Dasmunit_char_out=0x200
 test_hexout.o ../toollib.a ../../../../asmunit/asmunit.a -o test_hexout.bin
../../../../asmunit/asmunit_wrapper.sh test_hexout.bin 1000
Tests run: 6, Failures: 0
rm test_hexout.bin test_hexout.o

Py65 Monitor

       PC  AC XR YR SP NV-BDIZC
65C02: 0000 00 00 00 ff 00110000
.Wrote +504 bytes from $1000 to $11f7

       PC  AC XR YR SP NV-BDIZC
65C02: 0000 00 00 00 ff 00110000
.
[hexout]
PASS
PASS
PASS
PASS
PASS
PASS
       PC  AC XR YR SP NV-BDIZC
65C02: 1051 9f 00 00 ff 10110000
.
```

ASM, Tests und CI

- Was noch?
  - … ein paar Macros für die Tests und Mocks...
    - cmp16, cmp32 – vgl. 16/32 Bit-Wert an Adresse
    - set16, set32 – setze 16/32 Bit Wert an Adresse

FAT32 mit Tests und CI

- FAT32 - Los geht's, aber bitte „Test Driven“
  - bestehenden Code mit Tests absichern
  - „red“, „green“, „refactor“
    - Tests für neue Funktionalität schreiben
    - Code für neue Funktionalität schreiben

```
      ;  read n blocks from file denoted by the given FD and maintains FD.offset
      ;in:
      ;  X - offset into fd_area
      ;  Y - number of blocks to read at once - !!!NOTE!!! it's currently limited to $ff
      ;  read_blkptr - address where the data of the read blocks should be stored
      ;out:
      ;  Z=1 on success (A=0), Z=0 and A=error code otherwise
      ;  Y - number of blocks which where successfully read
fat_fread:
      jsr __fat_isOpen
      bne @_l_read_start
      lda #EINVAL
      rts
@_l_read_start:
      sty krn_tmp3                                  ; safe requested block number
      stz krn_tmp2                                  ; init counter
@_l_read_loop:
      debug "lp"
      ldy krn_tmp2
      cpy krn_tmp3
      beq @l_exit_ok

      lda fd_area+F32_fd::offset,x
      cmp volumeID+VolumeID::BPB + BPB::SecPerClus  ; last block of cluster reached?
      bne @_l_read                                  ; no, go on reading...

      copypointer read_blkptr, krn_ptr1             ; backup read_blkptr
      jsr __fat_read_cluster_block_and_select       ; read fat block of the current cluster
      bne @l_exit_err                               ; read error...
      bcs @l_exit                                   ; EOC reached? return ok, and block counter
      jsr __fat_next_cln                            ; select next cluster
      stz fd_area+F32_fd::offset+0,x                ; and reset offset within cluster
      copypointer krn_ptr1, read_blkptr             ; restore read_blkptr

@_l_read:
      jsr __calc_lba_addr
      jsr __fat_read_block
      bne @l_exit_err
      inc read_blkptr+1                             ; read address + $0200 (block size)
      inc read_blkptr+1
      inc fd_area+F32_fd::offset+0,x                ; inc block counter
      inc krn_tmp2
      bra @_l_read_loop
@l_exit:
      ldy krn_tmp2
@l_exit_ok:
      lda #EOK                                      ; A=0 (EOK)
@l_exit_err:
      rts
```

```
; --------------------
        setup "fat_fread 0 blocks 1sec/cl"
        ldx #(1*FD_Entry_Size)
        SetVector data_read, read_blkptr
        ldy #0
        jsr fat_fread
        assertZero 1
        assertA EOK
        assertX (1*FD_Entry_Size)
        assertY 0               ; nothing read

; --------------------
        setup "fat_fread 1 blocks 1/1"
        SetVector data_read, read_blkptr
        ldy #1
        ldx #(1*FD_Entry_Size)
        jsr fat_fread
        assertZero 1
        assertA EOK
        assertX (1*FD_Entry_Size)
        assertY 1
        assert32 $00006968, lba_addr ; expect $67fe + (clnr * sec/cl) => $67fe + $016a * 1= $6968
        assert16 data_read+$0200, read_blkptr
        assert32 $16a, fd_area+(1*FD_Entry_Size)+F32_fd::CurrentCluster
        assert8 1, fd_area+(1*FD_Entry_Size)+F32_fd::offset

; --------------------
        setup "fat_fread 2 blocks 2/1"
        SetVector data_read, read_blkptr
        ldy #2   ; 2 blocks
        ldx #(1*FD_Entry_Size)
        jsr fat_fread
        assertZero 1
        assertA EOK
        assertX (1*FD_Entry_Size)
        assertY 2
        assert32 $00006969, lba_addr ; expect $67fe + (clnr * sec/cl) => $67fe + $016b * 1= $6969
        assert32 $16b, fd_area+(1*FD_Entry_Size)+F32_fd::CurrentCluster
        assert16 data_read+$0400, read_blkptr ; expect read_ptr was increased 2blocks, means 4*$100
        assert8 1, fd_area+(1*FD_Entry_Size)+F32_fd::offset+0 ; still offset 1, we have a 1 sec/cl fat geometry
```

FAT32 mit Tests und CI

- Was könnten wir noch brauchen?
  - nicht-funktionale Tests
    - assertCycles <threshold> - vgl. verbrauchte Zyklen mit <threshold>

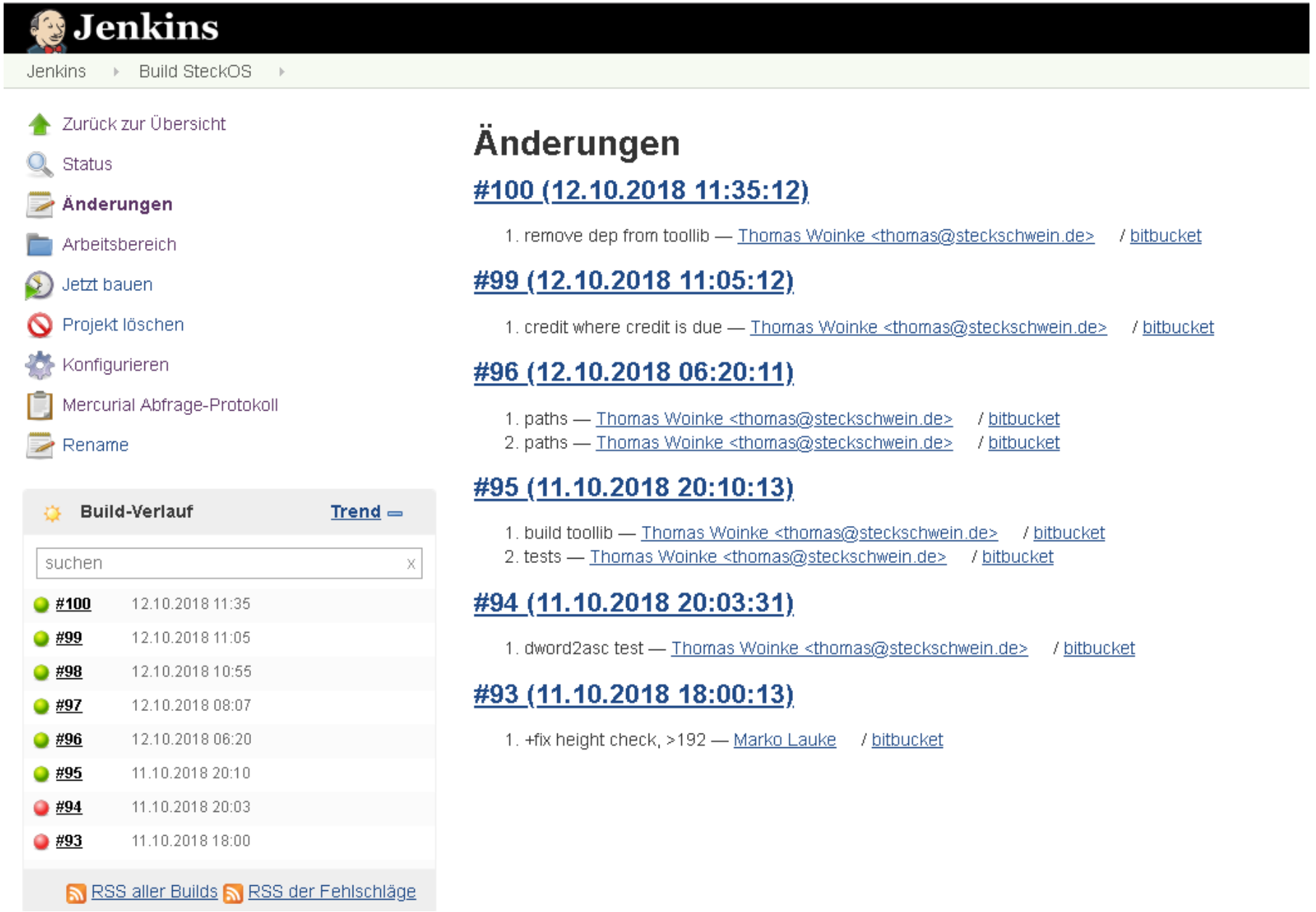
FAT32 mit Tests und CI

- Die Welt will eine 6502 FAT32-Implementierung,
  - die spec.-konform ist
    - wird durch die Tests belegt
  - die mit allen Varianten/Geometrien „klar kommt“
    - automatisches erzeugen von FAT32 Images und Mocks
  - die performant ist
    - wir testen auch nicht-funktional
  - „self contained“ - braucht nur read/write block
    - der Code ist „testbar“ und damit entkoppelt

… to be continued!

FAT32 mit Tests und CI

- Wieso eigentlich Continuous Integration (CI)?



Steckschwein mit Tests und CI

- Wie machen wir weiter?
  - in Software - fertige steckOS Builds aus Jenkins
    - als TAR-Ball, mit steckOS, mit allen Tools
    - ready to „steck“ 'n' play...
  - in Hardware - Einplatiner mit steckOS